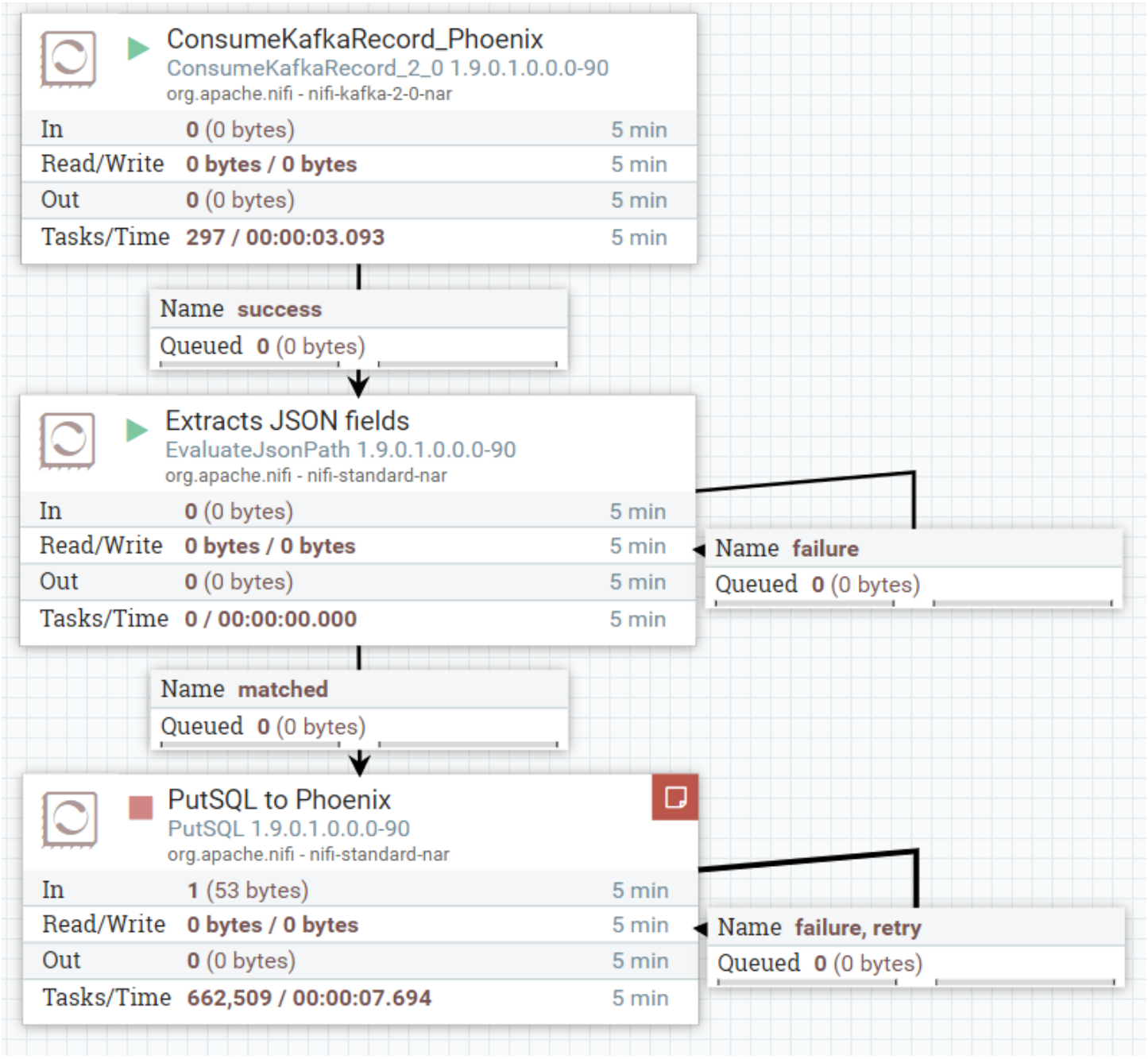
CDF Workshop Extra Labs

Extra Lab - Streaming to HBase/Phoenix

In this extra lab, we'll also send messages to HBase via Phoenix. Below is the flow we will build:



We start with a new **ConsumeKafkaRecord** processor. Copy and paste an existing one from the canvas and update the GroupID, so that each Consumer receives all messages.

Property	Value
Kafka Brokers	fabiog-one-1.c.gcp-se.internal:9092
Topic Name(s)	iot
Topic Name Format	names
Record Reader	reader_avro_iot →
Record Writer	writer_json_phoenix →
Honor Transactions	true
Security Protocol	PLAINTEXT
Kerberos Credentials Service	No value set
Kerberos Service Name	No value set
Kerberos Principal	No value set
Kerberos Keytab	No value set
SSL Context Service	No value set
Group ID	iot-phoenix

Also, create a new RecordWriter of type JsonRecordSetWriter and call it *writer_json_phoenix*. Configure it as below, ensure you set “One Line Per Object” in Output Grouping.

Property	Value
Schema Write Strategy	Do Not Write Schema
Schema Cache	No value set
Schema Access Strategy	Use 'Schema Name' Property
Schema Registry	Cloudera Schema Registry →
Schema Name	IoT-Schema
Schema Version	No value set
Schema Branch	No value set
Schema Text	${avro.schema}
Date Format	No value set
Time Format	No value set
Timestamp Format	No value set
Pretty Print JSON	false
Suppress Null Values	Never Suppress
Output Grouping	One Line Per Object

From here we connect to a **EvaluateJsonPath** processor to extract the relevant json fields. In this case, all of them. We are basically extracting the fields and put them as file **attributes**.

Property	Value
Destination	flowfile-attribute
Return Type	auto-detect
Path Not Found Behavior	ignore
Null Value Representation	empty string
id	$.id
ts	$.ts
val	$.val
val2	$.val2

Check any **Flowfile** after it goes through this processor: you'll see the exact same value in the Attributes page:

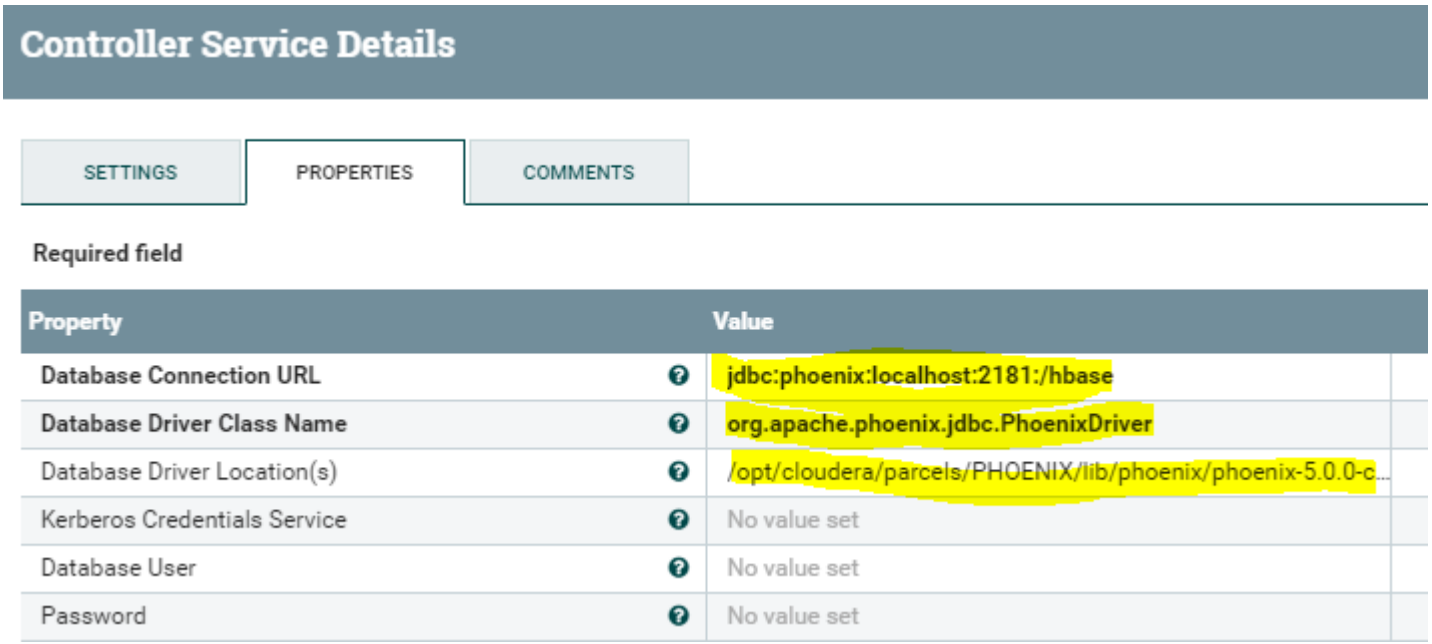
Now it's time to add the processor to put the data into Phoenix. Remember that Phoenix acts like an RDBMS on top of HBase, and as such we can use a standard JDBC connection to connect to it. Add a **PutSQL** processor to the canvas and open the Properties page.

SQL Statement: UPSERT INTO sensors VALUES (${id}, ${val}, ${val2}, ${ts:divide(1000)})

Quick Note: as you can see we divide the timestamp value by 1000: check the timestamp from above FlowFile, 1565714871718000. This is the epoch time in microseconds. We created this value in the GenerateFlowFIle and we need microseconds because Kudu stores the Timestamp in microseconds. HBase/Phoenix however store Timestamp in milliseconds, so we must divide the value by 1000.

For **JDBC Connection Pool,** create a new Service of Type 'DBCPConnectionPool' and conveniently call it *JDBC 2 Phoenix*, or something like that.

Then configure the Controller as follows:



Database Connection URL: jdbc:phoenix:localhost:2181:/hbase
Database Driver Class Name: org.apache.phoenix.jdbc.PhoenixDriver
Database Driver Location: /opt/cloudera/parcels/PHOENIX/lib/phoenix/phoenix-5.0.0-cdh6.2.0-client.jar

Save and Enable.

At this point your flow is complete, but you haven't created the Phoenix table yet! SSH into the host and become root. Then you can open the phoenix shell...

```
$ phoenix-sqlline localhost:2181
```

...and issue the SQL create command:

```
0: jdbc:phoenix:localhost:2181> CREATE TABLE IF NOT EXISTS SENSORS (ID
UNSIGNED_SMALLINT PRIMARY KEY, VAL UNSIGNED_SMALLINT, VAL2 UNSIGNED_SMALLINT, TS
TIMESTAMP);
```

Once you start the flow, query the table and you should see the data:

```
0: jdbc:phoenix:localhost:2181> select * from sensors;
```

```
+-------+------+-------+-------------------------+
|  ID   | VAL  | VAL2  |           TS            |
+-------+------+-------+-------------------------+
| 422   | 122  | 61    | 2019-08-19 13:28:59.526 |
| 1449  | 171  | 100   | 2019-08-19 13:29:09.531 |
| 2193  | 56   | 100   | 2019-08-19 13:28:39.523 |
| 2497  | 193  | 100   | 2019-08-19 13:29:09.531 |
| 2553  | 246  | 100   | 2019-08-19 13:28:19.517 |
| 3353  | 147  | 70    | 2019-08-19 13:28:19.517 |
| 4659  | 98   | 100   | 2019-08-19 13:28:49.525 |
| 4794  | 17   | 100   | 2019-08-19 13:28:29.521 |
| 4902  | 199  | 100   | 2019-08-19 13:28:59.526 |
| 5872  | 59   | 200   | 2019-08-19 13:28:49.524 |
| 7508  | 28   | 15    | 2019-08-19 13:29:19.535 |
| 7708  | 154  | 44    | 2019-08-19 13:28:29.521 |
| 9052  | 243  | 100   | 2019-08-19 13:29:19.535 |
| 9409  | 112  | 213   | 2019-08-19 13:28:39.523 |
+-------+------+-------+-------------------------+
14 rows selected (0.059 seconds)
```

This concludes the flow, congrats, well done!

Extra Lab - Streaming to Kudu via Spark Streaming and Kafka

Review the streaming pipeline

We will now configure a new pipeline to store streaming data into CDH for data analytics. The pipeline looks like follows: **Mock-data-generator → Kafka → Spark Streaming → Kudu → Impala**

- A streaming data generator simulates FX transactions.
- These transactions are sent to a Kafka topic.
- Spark Streaming will read from the topic, process the data and insert into Kudu.
- Impala will read and analyze the data in realtime from Kudu.

Download and configure the mock data generator

```
$ cd ~
$ wget https://s3-us-west-2.amazonaws.com/cloudera-public-data/partners-workshops/mock-data-generator.tar.gz
$ tar zxvf mock-data-generator.tar.gz
```

Also download some required libraries

```
$ wget  http://central.maven.org/maven2/org/apache/kudu/kudu-spark2_2.11/1.9.0/kudu-spark2_2.11-1.9.0.jar
$ wget https://raw.githubusercontent.com/swordsmanliu/SparkStreamingHbase/master/lib/spark-core_2.11-1.5.2.logging.jar
```

Run the jar file to output the simulated trading transactions to stdout to familiarize yourself with the data it is creating.

```
$ sed -i "s/localhost/$(hostname)/" mock-data-generator/conf/kafka.config
$ sed -i "s/localhost/$(hostname)/" mock-data-generator/sparkstreamingkudu.py
$ java -jar mock-data-generator/mock-data-generator.jar logger.config
[...]
2019-04-09 15:31:48,851 DEBUG n.a.d.j.g.l.KafkaLogger [Thread-1] Sending event to Kafka: [ {
      "tradeid":"d7f19299-be80-45a8-929b-89300f21bbe3",
      "tradedate":"2019-04-09T15:31:48.851Z",
      "venue":"FXALL",
      "cpty":"Bulldogs Inc",
      "dir":"S",
      "ccypair":"EUR/USD",
```

```
      "Amt":"1000000",
      "Price":1.1826,
      "valuedate":"2019-04-11T15:31:48.851Z"
} ]
```

The generator outputs a tab separated line of text. Each value represent in order:

1. Trade ID
2. Trade Date
3. Venue
4. Counterparty
5. Direction (Buy or Sell)
6. Currency Pair
7. Amount
8. Price
9. Value Date

We are receiving some transaction data such as the amount of the deal, the price and the traded currency pair. Some of those amounts are in USD, but some others are expressed in Euro or Pound Sterling. If we want to record the USD equivalent amount for every transaction, we need to do some simple math, multiplying the amount by the price. We only need to do so if the term currency of the traded pair is USD, as in EUR/USD, but not in USD/CAD. We'll use spark streaming for this data processing.

Stop the process by pressing **Control + C** on your keyboard.

Later, to send data to the Kafka topic, you will run the app with a new parameter, as below:

```
$ java -jar mock-data-generator.jar kafka.config
```

Create Kafka topic

Create the **Kafka topic** called '**t**' with the command line below. As we are running on a single node cluster, we have to limit the replication factor to 1. Ideally, for High Availability in production, you would set at least 3.

```
$ kafka-topics --create --zookeeper `hostname`:2181 --replication-factor 1 --
partitions 1 --topic t
```

Output:

```
[...]
18/10/12 07:53:10 INFO admin.AdminUtils$: Topic creation
{"version":1,"partitions":{"0":[36]}}
Created topic "t".
18/10/12 07:53:10 INFO zkclient.ZkEventThread: Terminate ZkClient event thread.
```

```
18/10/12 07:53:10 INFO zookeeper.ClientCnxn: EventThread shut down
18/10/12 07:53:10 INFO zookeeper.ZooKeeper: Session: 0x166689eb7c0006a closed
[cloudera@quickstart ~]$
```

Check the **Appendix** for commands to manually send and receive messages, in case you want to test Kafka out to familiarize yourself.

Create the Kudu Table for Streaming Trade Data

Create the **Kudu table**:

```
CREATE TABLE t
(
 tradeid STRING,
 tradedate STRING,
 venue STRING,
 cpty STRING,
 dir STRING,
 ccypair STRING,
 amt DOUBLE,
 price DOUBLE,
 valuedate STRING,
 usd_amt DOUBLE,
 PRIMARY KEY(tradeid)
)
PARTITION BY HASH PARTITIONS 2
STORED AS KUDU
TBLPROPERTIES (
  'kudu.master_addresses' = 'localhost:7051',
  'kudu.num_tablet_replicas' = '1'
);
```

Run the simulation

Now that we have the pieces of the puzzle all configured, it's time to run the simulation.

1. Start the data-generator to send data to the Kafka topic 't'

```
$ java -jar mock-data-generator/mock-data-generator.jar kafka.config
```

Optionally, you could open a new terminal window and subscribe to the kafka topic t.

```
$ kafka-console-consumer  --bootstrap-server `hostname`:9092 --topic t
```

2. Copy or move file **sparkstreamingkudu.py** file that came inside the mock-data-generator package to your home folder. Remove the partial mvn project which will be fully reloaded as part of our spark job with all files needed.

```
$ mv ~/mock-data-generator/sparkstreamingkudu.py ~
$ rm -rf ~/.m2 ~/.ivy2/
```

Now you can run the spark streaming job

```
$ spark-submit --master local[2] --jars kudu-spark2_2.11-1.9.0.jar,spark-
core_2.11-1.5.2.logging.jar --packages org.apache.spark:spark-streaming-
kafka_2.11:1.6.3 sparkstreamingkudu.py
```

Inspect the code in file *sparkstreamingkudu.py*: can you follow what it does? VIM is installed in your sandbox environment, take a look at it using vim so you've syntax highlighting.

3. Query Kudu to see that spark streaming calculated the USD amount..

```
select tradeid, ccypair, amt, price, usd_amt from t limit 5;
```

```
+--------------------------------------+---------+---------+--------+---------+
| tradeid                              | ccypair | amt     | price  | usd_amt |
+--------------------------------------+---------+---------+--------+---------+
| 0d2c84a2-2d96-4fde-b7a6-9b08f4311069 | GBP/USD | 1000000 | 1.3042 | 1304200 |
| 39e58ca7-6346-4f5e-a909-7926b55263a9 | GBP/USD | 2000000 | 1.3061 | 2612200 |
| 45370dee-5184-4d52-af78-44afeead292f | GBP/USD | 500000  | 1.3052 | 652600  |
| 5cc00aa0-390f-4748-b2af-f6144fe4f721 | EUR/USD | 2000000 | 1.1782 | 2356400 |
| 5e037514-11d4-41e5-b421-808bf373336d | GBP/USD | 1000000 | 1.3178 | 1317800 |
+--------------------------------------+---------+---------+--------+---------+
Fetched 5 row(s) in 0.22s
```

You can now stop both the mock-data-generator and the spark streaming apps with Ctrl+C.

APPENDIX

References

Script to recreate the lab environment.

Install Kudu using Cloudera Manager documentation (packages or parcels).

Official Apache Kudu and Apache NiFi websites.

Impala DataTypes

Mock data generator GitHub project website.

Kafka Commands

```
# Topics: create/delete/list/describe
$ kafka-topics --create --zookeeper <zk-hostname>:2181 --replication-factor 1 --
partitions 1 --topic t

$ kafka-topics --describe --zookeeper <zk-hostname>:2181 --topic t

$ kafka-topics -zookeeper <zk-hostname>:2181 --delete --topic t

$ kafka-topics -zookeeper <zk-hostname>:2181 --list


# subscribe to topic from broker servers
$ kafka-console-consumer  --bootstrap-server <kafka-broker-hostname>:9092 --from-
beginning  --topic t

# subscribe to topic from zookeeper
$ kafka-console-consumer --zookeeper <zk-hostname>:2181 --topic t --consumer-
property group.id=hello

# publish to topic manually from stdin
$ kafka-console-producer --broker-list <kafka-broker-hostname>:9092 --topic t
```